Object Mapping and Detection Using Ultrasonic Range Finding with DE2 Bots

Philip Balê
Caitlyn Caggia
Eydan Fishel
Can Kabuloğlu

ECE 2031, Digital Design Lab
Section GTL

Georgia Institute of Technology
School of Electrical and Computer Engineering

Submitted
April 7, 2016

Executive Summary

This design proposal presents a method to detect and locate objects using ultrasonic rangefinding with a DE2Bot. The purpose of this project was to improve the sonar mapping capability of the DE2Bots. The robot will navigate through an arena, represented as a 4 x 5 grid, with no more than 6 objects randomly placed outside of the center aisle. First the existing functionality of the DE2Bot will be explored, including movement, ultrasonic range finding, and radio communication. A depth-first search algorithm will be developed and tested in Python to ensure that all objects will be found in all possible cases with only one object per tile. Then this algorithm will be converted to assembly code for implementation with a DE2Bot. Additional assembly code will be developed to move the robot along a clear path through the arena, detect and record objects and their locations, and transmit data containing object count and coordinates. If the depth-first algorithm is not able to be implemented on the SCOMP, a back-up search algorithm is outlined in a contingency plan. A Gantt Chart is included to provide a project plan and timeline.

Object Mapping and Detection Using Ultrasonic Range Finding with DE2 Bots

Introduction

This document proposes a method of sonar mapping using a DE2Bot to count and locate objects in a walled rectangular arena. The team will first learn existing features of the DE2Bot, including how to move the robot, how to measure the distance of objects with the ultrasonic sensors, and how to relay coordinates via radio signal. The search algorithm uses a depth-first search that will first be tested using a Python simulator to ensure that the robot will find all objects in all possible cases, with no more than 6 objects total. Then, the algorithm will be converted to assembly code for use on a DE2Bot. Additional code will be developed to move the robot along a clear path through the arena and transmit coordinates containing object locations.

Technical Approach

The objective of this project is to program a DE2Bot to detect and locate no more than 6 randomly placed objects on the field, and send the number of objects and their locations to a central computer in 60 seconds or less.

The team will develop code to:

- Move the robot to specific locations on the field without hitting objects or walls
- Detect nearby objects using the ultrasonic sensor
- Record the location of detected objects in memory
- Determine the robot's next location using a search algorithm
- Transmit number of objects and object locations via radio signal.

Moving on the Field

First, the team must move the DE2Bot on the field accurately, keep track of the DE2Bot's position, and avoid collisions with objects or walls. Since all objects will be a placed on a grid, the DE2Bot should move only in straight lines or 90-degree turns. To navigate the arena, the team needs code to:

- Turn left and right at 90^o angles
- Move forward and backwards 2 feet (the length of one tile on the grid)
- Track the DE2Bot's location inside the arena
- Check the space before the DE2Bot to avoid crashes.

Each step will be independently coded in assembly as subroutines and tested on the field for accuracy; later these subroutines will be worked into the main search code. The subroutines for right angle turns and tracking the DE2Bot's location on the grid can be found in Appendix A.

The mechanics of the DE2Bot introduce odometry error when the robot moves forward, backward or turns. The error increases with the length of distance traveled, so the team chose to have the robot move only one tile at a time and rotate no more than 90 degrees at a time. Movement error is also reduced by selecting a wheel velocity high enough to make the wheels move, but low enough that inertia has minimal effects.

Detecting Objects

The DE2Bot will check on the left, right, and directly forward to see if an object is present in an adjacent tile. Sonar sensors 0 (left side), 5 (right side), 2 and 3 (front) will be used to collect the distance data in millimeters. If an object is less than 2 feet away (or 610 millimeters) in a given direction, that tile will be recorded as a location of an object. However, if the robot is currently next to a wall, the search algorithm will recognize that a detected surface is the wall and not an object and no location will be recorded. Since the robot will only be checking for objects immediately adjacent to its current tile, all surfaces will be perpendicular to the sensor, which reduces error in sonar measurements.

Recording Object Locations

The DE2Bot will send the coordinates of objects to a central computer via wireless serial communication. Location data will be kept directly in the DE2Bot's memory as three different variables (x-position, y-position, and object count), each stored as 16-bit words. Every time the robot detects a new object, object count will be incremented by one. Each bit of the 16-bit word will belong to a specific tile of the field grid, as there are 16 different possible locations of objects. When DE2Bot detects an object, the single bit referring to that specific tile will be changed from 0 (empty) to 1 (full).

Determining the Robot's Next Position

Object searching is performed by following a depth-first search path. To begin the search, the robot starts 2 feet outside of the arena, facing the inside, and moves forward until the robot is in the center of the first tile. The search pattern starts once inside the first tile. First, all neighboring grid cells are examined. If a new object is detected, it is stored in memory. Next, possible moves are determined for the directions right, forward, and left. If a move is available and has not been previously taken, it is then performed based on the ordering of right, forward, and left. Subsequently, neighboring grid cells are then examined (repeating the first step). If the search reaches the end of a path where no more moves are available, it then moves backwards until a move becomes available. When all possible grid cells have been examined or traversed, the search is complete.

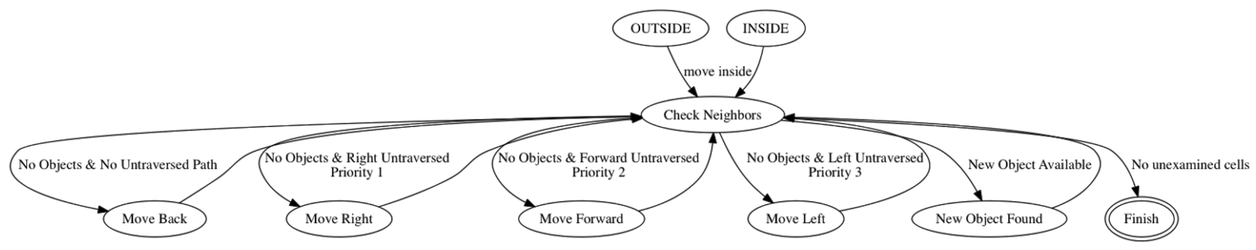
OUTSIDE
INSIDE
move inside
Check Neighbors
No Objects & No Untraversed Path
No Objects & Right Untraversed Priority 1
No Objects & Forward Untraversed Priority 2
No Objects & Left Untraversed Priority 3
New Object Available
No unexamined cells
Move Back
Move Right
Move Forward
Move Left
New Object Found
Finish

Figure 1. Flow chart showing the decision making process used to determine where the robot will move next. Possible states are Check Neighbors, where the robot will detect any nearby objects with the sonar sensors, Move Back/Move Right/Move Forward/Move Left to control the robot's movement, New Object Found to record object locations, and finish to send coordinates with object locations.

Transmitting Object Count and Locations

The design manual suggests that it takes 200 ms to send a 10-bit word to the central computer, meaning that it should take less than a second (around 640 ms) to send both the number and location variables. Code to send data from the DE2Bot to the central computer was provided with the project details. Data will be sent at the very end of the field search; if time constraints prove to be problematic, then data will be sent every time the robot finds a new object.

Management Plan

Appendix B contains a Gantt Chart outlining the intermediate tasks to complete the project and related reports and presentations. Major tasks include completing the final presentation, preparing for the demonstration, and writing the final report. To ensure a successful demonstration, time is reserved for learning the DE2Bot's features, testing a search algorithm with the Python simulator, and implementing the search algorithm for use with the DE2Bot.

<u>Contingency Plan</u>

All of the tasks in the Gantt Chart should be feasible to complete within the timeline of this project. If the depth-first-search algorithm in Python uses too much recursion to implement in SCOMP, iteration could be used to fix the stack overflow problem. Two of the tasks, coding and testing the DE2Bot's movement and sonar capabilities and implementing a search algorithm, were allocated the greatest amount of time. If these tasks are completed early, a more optimized movement function (like turning faster or looking more than one block away at a time) could be added to decrease the total time for the DE2Bot to detect and locate all of the objects in the arena.

Appendix A: Assembly Code for DE2Bot Movement and Location Tracking

```
; MainRobotCode.asm (excerpt)
; Sample movement subroutines for DE2Bot
; Team I (The Metz)
; Georgia Institute of Technology - ECE 2031
; April 7th 2016


;;; SUBROUTINE TO TURN RIGHT ;;;

RightTurn:                   ; Subroutine to turn right 90 degrees
      IN      THETA          ; Read robot orientation
      ADDI    -90            ; Decide if less than 90
      JPOS    Rn1            ; If greater than 90 ...
      JZERO   Rn1            ; If equal to 90 ...
      JUMP    Rr1            ; If less than 90 ...

Rn1:                         ; Condition of having a theta >= 90
      IN      THETA          ; Read orientation again
      ADDI    -90            ; Determine the target orientation
      STORE   target         ; Save it to target variable

      LOAD    FSlow          ; Out the necessary velocities
      OUT     LVELCMD
      LOAD    RSlow
      OUT     RVELCMD

Rn2:
      IN      THETA          ; Loop, condition of having a theta >=
90
      SUB     target
      JPOS    Rn2            ; Loop until it reaches the target
      JZERO   EndTurnR       ; If reached target, jump to finish turn
      JUMP    EndTurnR       ; If reached target, jump to finish turn

Rr1:                         ; Condition of having a theta < 90
      IN      THETA          ; Read orientation again
      ADDI    270
      STORE   target         ; Determine the target orientation

      LOAD    FSlow          ; Out the necessary velocities
      OUT     LVELCMD
      LOAD    RSlow
      OUT     RVELCMD

Rr2:                         ; Loop, condition of having a theta < 90
      IN      THETA
      ADDI    360            ; Add 360 to work with 0-359 angles
      SUB     target
      JPOS    Rr2             ; Loop until it reaches the target
      JZERO   EndTurnR         ; If reach target, jump to finish turn
      JUMP    EndTurnR         ; If reached target, jump to finish
turn

EndTurnR:
      LOAD    Zero             ; Stop the robot by outing 0 to motors
      OUT     LVELCMD
      OUT     RVELCMD
      STORE   target           ; Make target 0 again for next use

target: DW      0              ; Initiate target variable

;;; SUBROUTINE TO MOVE FORWARD ;;;

moveForward:
       IN      RPOS            ; Load the RPOS
       ADD     Tile            ; Add the length of a tile
       STORE   rTarget         ; Save as the right wheel target

       IN      LPOS            ; Load the LPOS
       ADD     Tile            ; Add the length of a tile
       STORE   lTarget         ; Save as the left wheel target

       IN      THETA           ; Load the current orientation angle
       STORE   oldOrient       ; Save it to keep orient the same

loopMove:
       LOAD    FMid            ; Load the velocity value
       OUT     RVELCMD         ; Out it to right wheel
       OUT     LVELCMD         ; Out it to left wheel

       IN      RPOS            ; Load RPOS
       SUB     rTarget         ; Check if right wheel is at target
       JNEG    loopMove        ; If not loop again

       IN      LPOS            ; Load LPOS
       SUB     lTarget         ; Check if left wheel is at target
       JNEG    loopMove        ; If not loop again

       LOAD    Zero            ; Target is reached, load zero
       OUT     RVELCMD         ; Out it to right wheel
       OUT     LVELCMD         ; Out it to left wheel

       RETURN                  ; Return to caller
```

```
                                ; Define the required variables
rTarget:    DW      0
lTarget:    DW      0
oldOrient:  DW      0

;;; SUBROUTINE TO GET X-AXIS COORDINATE ;;;

getY:                            ; Subroutine to obtain Y-axis coord.
        LOAD    Zero             ; Load zero
        STORE   Result           ; Reset the result variable
        IN      YPOS             ; Load the current y position
getY2:                           ; Loop to do the division operation
        SUB     Tile             ; Subtract the lenght of one tile
        STORE   tempY            ; Save the remainder
        JNEG    returnY          ; If division is complete, return
        LOAD    Result
        ADD     One              ; If not, increment the quotient by 1
        JUMP    GetY2            ; Loop again to finish division

returnY:
        LOAD    Result           ; Bring the result to the AC
        RETURN                   ; Return to caller


Tile:   DW      581              ; Save the value of the tile length
Result: DW      0                ; Initiate the result/quotient var.
TempY:  DW      0                ; Initiate the temp remainder var.
```

Appendix B: Gantt Chart